Software Interface Analysis Tool (SIAT) Architecture Definition Document (NASA Center Initiative)

DID 06
Contract NAS2-96024
December 30, 1997

Prepared for:

NASA Ames Research Center
Moffett Field, CA 94035-1000

Prepared by:

Intermetrics, Inc.
NASA Software IV&V Facility
100 University Drive
Fairmont, WV 26554

TABLE OF CONTENTS

LIST OF EXHIBITS

1. Introduction

Many of the large software systems being built today are highly distributed, with complex interactions between the software components. Interfaces between software components are particularly hard to verify for correctness, and are therefore a major source of software hazard. Interface errors have been implicated in the majority of software failures, especially those in large, safety-critical applications[1]. Such errors have a number of causes, related to the difficulty of coordinating many separate developer organizations working in parallel. The interfaces between components written by different organizations are defined in Interface Control Documents (ICDs). However, it is a complex task to ensure that each component is consistent with the interface definitions, and that the changes that occur as the designs evolve are fully propagated to all the components that are affected. Each software component may support a multitude of interfaces, and there may be large variations in the development maturity across different components and development organizations. Even with the current best practices in the software industry, many interface errors are not detected until system integration or later.

Existing tools provide very little support for verifying that interactions between components in the as-built system conform to the specification. A major weakness is in the ability to trace dataflows across subsystem boundaries. Verifying that the sender is providing the right data at the correct rates, units, and format that the receiver expects and needs is often a difficult and tedious process. Weaknesses in the ability to specify and manage software interfaces, together with the inherent difficulty in verifying external interfaces during the code and unit test phase often results in complications during system integration and test. When problems proliferate to this point, cost, schedule, and mission are jeopardized.

The long-term objective is to build the tool infrastructure necessary for performing V&V on the new generation of complex, distributed systems. As software systems get larger and even more complex, dependence on manual inspection as the means for verifying software interfaces and determining impacts on proposed changes must be minimized. The ability to adequately perform V&V will depend on the level of tool support that is available to assist the V&V practitioner in efficiently locating and verifying interface data and the related data flow. The primary objective of this research effort was to identify and prototype new tool capabilities to support the interface analysis process, by:

〈 extending existing code navigation and browsing facilities to encompass the tracing of dataflow, data use, and data dependencies between Computer Software Configuration Items (CSCIs), and

〈 developing new techniques for performing consistency analysis between the implemented CSCIs, the design models, the documented software specifications, and interface definitions in the ICDs.

2. Architecture Overview

The SIAT architecture detailed in this and the following sections details a Graphical User Interface (GUI) solution that is web based in support of geographically dispersed users. Exhibit 2-1 depicts the proposed SIAT phase 2 architecture.

[1] Leveson, N. “Safeware: System Safety and Computers” Addison Wesley, 1995, p143

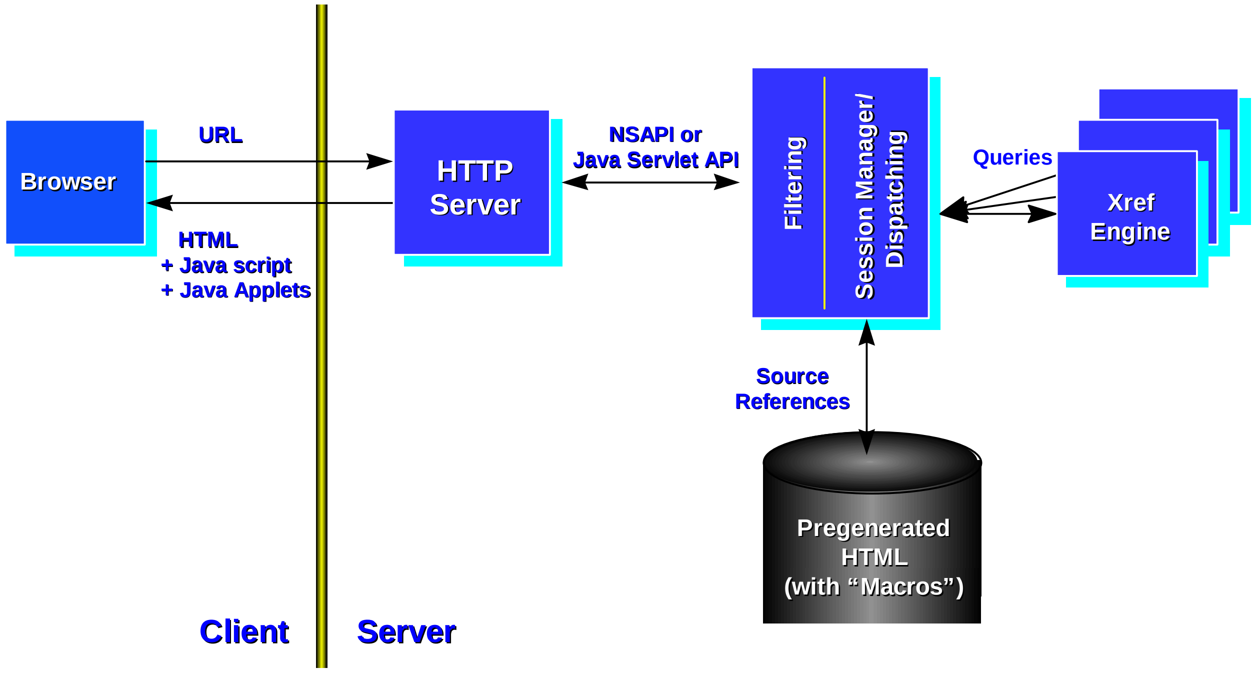

Exhibit 2-1 SIAT Phase 2 Architecture

State information associated with an analyst's session is saved on the server not the client. On the client, the browser stores a "cookie" to associate a particular user's session with the saved state on the server. The various parts of the architecture are discussed in more detail below.

The current implementation of the tool has an emacs interface. When the user edits a file with an Ada extension, it starts up the adam_browser (AdaMagic Browser) process. This process handles the queries from the user by looking at data in the Ada program library. In order to prevent confusion between the web browser on the client and the AdaMagic Browser on the server, the AdaMagic Browser is referred to as the cross-reference (or Xref) engine in this document. One advantage of this architecture is that the same Xref engine can be used as the back end of both the web and the emacs interface. It simply requires the development of an interface that is compatible with both.

3. Client Architecture

A standard web browser provides the framework for the client. A modern browser that supports Java, Java script, Cascading Style Sheets (CSS level 1) and Dynamic HTML will be required. Both Netscape Navigator (version 4.0 or higher) and Internet Explorer (version 4.0 or higher) meet these criteria. While there may be a substantial number of earlier versions of the two browsers in use currently, by the time the tool is complete the latest versions will be in widespread use, and other browsers will also support the required functionality.

As shown in Exhibit 3-1, the user can have almost any client configuration, with the supported browsers being available on PC, Macintosh, and Unix platforms. The server the clients use could be on their local area network, but could also be external to their network on the Internet. There would be no difference in the interface functions, but the user might see a difference in response time depending on their Internet connection. Even if a user has a slower Internet connection, the response should be acceptable as most of the content to be shown is text (Ada source files and query results) and not graphics which tend to take longer to download.

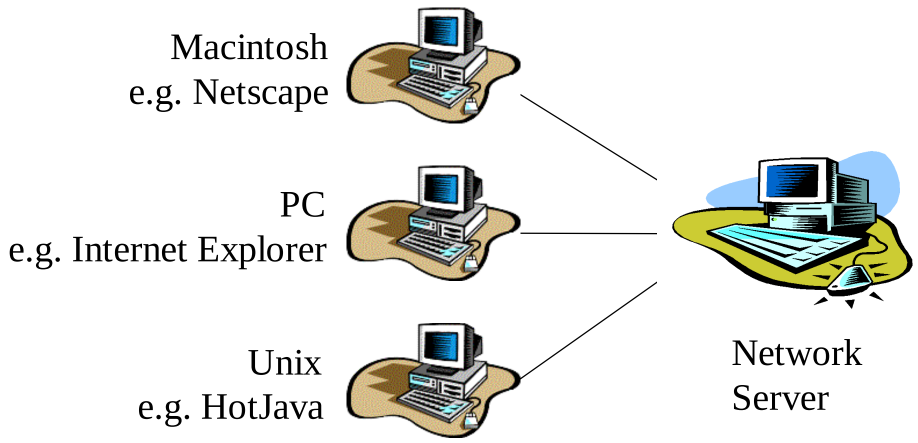

Exhibit 3-1 Client Configuration

In general, sequential browser requests from a user are completely independent of each other. The server simply returns the pages that are requested regardless of which browser made the request, so there is nothing to distinguish one user from another. In this case an analyst will have a context in which they are working, and all requests will be made within that context. The context is the Ada libraries that they are browsing. In order to distinguish one user from another the system will generate a unique session id for each user that will be stored as a "cookie" in the browser. When there is a cookie associated with a specific site, all requests that go to that site from the browser include the cookie as part of the request that is sent. On the server the session id will be mapped to the context for that user.

As previously mentioned, most of the content will simply be HTML formatted text. The main exception corresponds to the various kinds of graphs that can be generated (call graphs, dependency graphs, type hierarchies, data flow graphs). While a graph can be represented as an indented list as is done in the emacs version of the tool, it would be much more intuitive to the user if drawn as a directed graph. For this reason a Java applet will be used to render the directed graphs. The Java applet should allow the user to pan and zoom, and to select a particular node in the graph to bring up the source view of that node. In this case the content being shipped to the client is still not graphics; it is a Java applet and the data to render. The first time the user views a graph, the response will probably be noticeably slower because the applet is being downloaded along with the data. After that the applet will be in the applet cache on their client, and only the data will be downloaded.

3.1 Browser Dependencies

Our goal will be to minimize browser dependencies. The WorldWide Web Consortium (W3C) has a number of committees dedicated to standardizing the protocols and formats used on the web. At any given point, however, the browser manufacturers have added new features that may or may not become part of a standard in the future. For the most part the tool depends on formats and features that have been published as standards, primarily HTML and CSS1. There is also a standard for the scripting language, called ECMAscript, and the two implementations, Java script and Jscript are mostly compatible with it, but care will still be necessary when creating the scripts.

In the areas where the technology is still new and the standards are just emerging, notably Dynamic HTML, new features will only be used where they provide considerable benefit, such as the ability to have pop-up menus. Since there is already a committee at W3C that is working on the Document Object Model, a more consistent implementation of Dynamic HTML can be expected probably within the next year. For now, browser dependencies will be isolated, and different versions of the scripts will be implemented as necessary for the two browsers.
While Java has been touted as the answer to being able to run on all platforms, our experience has shown otherwise. While things are improving all the time, there are differences between the virtual machine implementations, and the bugs that show up in the different implementations.
Java Bean technology promises to provide reusable components for use in other applications, but most beans available today are still quite new and immature. Once again, use of this technology will be limited to places where it provides considerable benefit that outweighs the risk of immature components.

4. Client Server Interface

All browser requests are Universal Resource Locator (URL) requests using the HTTP protocol. The URL format can be broken down into the portion that identifies the server's address and the portion that identifies the page to return from that server. Many requests from the user will be for Ada source listings. The URL to request a source listing will simply refer to the file to be returned. When the user queries the Ada library, the query will be encoded in the URL. For example, if the user wants to find all uses of the object X declared in the package Y, the query would need to identify the type of query, and the object. The URL for this request might look like:

http://server.com/siat/find_uses/all/Y/X

The server would parse the URL, extract the parameters, and call the cross-reference engine with the appropriate parameters. All queries can be encoded into URLs in this way.

The response from the server is always an HTML page. The page may have been completely generated by the server, or it may be a static page on the server's file system. Embedded in the page there may be Java applets, or references to Java script routines depending on the type of result. In the future there may be other objects embedded in the page as the browsers continue to extend their features. However, the basis of the response will always be an HTML page that is displayed by the browser.

5. Server Architecture

5.1 Cross-Reference Engine

The xref engine uses the listing and information files generated by the Ada compiler to respond to queries from the user. Instead of developing a new xref engine for the web client, the same engine that is used by the emacs client will be used. This engine was not designed to be run over a network in a client/server architecture, it is designed to work with only one user, not multiple users. For this reason one xref engine process must be run for each user.

The xref engine maintains the state for the user. This includes the library context, and other session parameters such as the current search scope. In fact, the xref engine must maintain all state information that is needed for it to support the functions provided by either the web or emacs interfaces. The Session Manager handles the mapping between the user and their xref engine.

5.2 Session Manager/Dispatcher

When a user signs on the session manager assigns them a unique session id. The response from the server to the initial sign-on includes a cookie containing the session id. Other information about the user's session could also be embedded in the cookie, such as the information needed to resume a previous session. However, the first prototype will only implement the session id.

When the session is started the session manager starts up a new xref engine to handle this session's requests. The xref engine is associated with the session id so that all subsequent requests from the same user will be directed to the same xref engine. The xref engine will be shut down when the user ends their session with the **close library** command. Unfortunately, there will always be times when the user forgets to end their session before browsing to another page, or loses their connection to the server for some reason. To be able to cope with these situations, there must be a time-out period for shutting down the xref engine, and invalidating the session. It will be implemented so that if there have been no requests from a particular session in some number of hours, then it is considered to be finished. The number of hours will be configurable.

When a query request comes in, the dispatcher will use the session id cookie to send the request to the appropriate xref engine. The response from the xref engine will be reformatted to display in the browser. Since the xref engine will be the same as the one used by the emacs interface, these responses will not have the HTML tags already embedded in them. The text responses follow a format that should be easily converted to HTML. If processing the responses takes too much time, an alternate approach would be to implement a switch on the xref engine to return HTML instead of plain text.

If the request is for a source listing instead of a query, the source file will be retrieved and passed through the filtering phase before returning it to the user.

5.3 Filtering and Macros

The HTML formatted Ada source listings will be generated when the software component is installed on the system. The source listings could be generated on the fly as a request is received, but as the sources are fairly static for this application it will provide better performance to have them preexisting.

The pregenerated listings will use standard HTML for most of the formatting. There are some parts of the interface that are client dependent, or that may change over time as additional features are supported. The pop-up menus are a good example of something that is both client dependent and likely to change in the future. To isolate these changeable pieces of the interface, a set of macros will be defined. When a listing that contains macros is to be returned to the user, a filtering step is run to expand the macros.

The main benefit of the macros is that the sources can be generated once, and not changed by the implementation of new features. When new features are added to the macros themselves, the filtering step will automatically pick up the enhancements without any changes to the source listings. As the Document Object Model becomes standardized it should be possible to update the scripts to use it without regenerating the sources. In the future the look of the client can be changed by changing the macros.

Another benefit of the macros is that they can be expanded with a version implemented for the user's specific browser. Many scripts that are browser independent include all implementations in the script, and the script can simply choose what browser dependent information to send by looking at the version of the browser. With the macros, the filtering process can look at the version of the browser, and simply return the right implementation. This may also reduce the amount of data that is transmitted to the client.

Finally, the same mechanism could be used to handle multiple languages. For example, assume that for all languages there is a macro to indicate a pop-up menu. The contents of the menu would be language dependent, and the style of the menu could also be language dependent. There would simply need to be a parameter to the filter to indicate which language to use as well as which client browser, then the correct menu implementation could be inserted. Alternatively, new macros could be added for new languages so that the language parameter would be unnecessary.

5.4 HTTP Server

While a Netscape server specific solution would suffice for phase 2, in the long term, the SIAT server should be able to run with all the most common HTTP servers. For this reason, a server independent solution is preferred for the prototype as well. Most servers implement an API to allow the addition of new services to the server. The session manager, dispatcher and filters are a set of new services related to V&V to be added to the server. For the prototype there are several choices for the API: Netscape Server API (NSAPI), Netscape Web Application Interface (WAI), and Java Servlets.

5.4.1 Server Interfaces

The NSAPI has been around the longest of the three and is the most stable. We have experience implementing interfaces using this API, and it would be the least risky approach. However, it would be a Netscape specific approach. If we decided to support the Apache server, or the Microsoft server, the server functions would need to be modified to use those servers' interfaces. Another disadvantage of this interface is that it is being phased out in favor of Netscape's new interface, the Web Application Interface.

The Netscape WAI is based on CORBA. This has the advantage that HTTP server and the application implementing the V&V services do not have to be on the same machine. The interface would be the same regardless of where the V&V services were running, and if necessary they could be on a different machine for load balancing. It is also the Netscape recommended way to implement new features because it will continue to be the interface that is supported by future servers. However, it is very new and immature. Industry reports indicate that it is not yet robust enough to implement a new service that requires high stability. It is also Netscape specific, so once again, the server functions would have to be modified to support other servers.

Java Servlets are another new interface to HTTP servers, and like the Netscape WAI suffer from immaturity. Servlets are intended to be for servers, what applets have become for browsers. The biggest advantage of servlets is that they are server independent. Javasoft has implemented the servlet API for the Netscape Enterprise Server, the Apache server, and the Microsoft server. They are also supported natively by Javasoft's Java server. One thing to note is that the Netscape servlet API currently uses the NSAPI, but they are working with Netscape to develop an interface based on the WAI. Another concern with the Java servlet interface is performance. The new services would be implemented in Java, an interpreted language, whereas, they would be implemented in C or C++ under the other approaches.

We plan to continue to investigate the servlet approach because it would be server independent. However, if we are not quickly convinced that it will be robust and fast enough to meet our needs, then we will use the NSAPI because it would be the least risky approach.

5.5 Source Repository

There will be a source repository on the server for all the software components that are installed. Within the source repository there will be a subdirectory for each CSCI that has been installed in the system. Within each CSCI there will be subdirectories for each version of the CSCI. All the sources, library files and listings for a particular version will be kept within the version's subdirectory. The directory structure of the sources may correspond to any directory structure used during development, or it may simply be a flat source directory. The standard subdirectories will be used for the library and generated listing files.

5.5.1 HTML Listing Generator

The Ada Lister already generates several different listings. The HTML listing will be an additional listing, and it will require the addition of another command line flag to request it. The HTML listings will contain HTML formatted source using cascading style sheets, along with embedded macros, as discussed above.

The listings will be generated when a new component is installed in the system. Storing the HTML listings will take additional disk space on the server, but response time will be faster when an analyst requests a particular source. Also, if several analysts are browsing the same source it is only generated once, not once each time someone requests it. If space becomes a concern, the decision not to generate the HTML listing on the fly could be revisited.

5.5.2 Source Repository Configuration File

When the user signs on to the system they are presented with a list of CSCIs and versions from which to choose. In order to present this list a configuration file listing the contents of the source repository must be maintained. Initially this configuration data will be kept in a plain text file. Each line in the file will correspond to one CSCI version that is in the repository. Tabs will separate the standard information about that CSCI version, such as the date that it was released. When a new CSCI is added to the system, this file will be updated with the necessary information as part of the installation process.

5.6 Persistent User Data

One of the features available to analysts is the ability to extract source constructs from a CSCI for later comparison and analysis. This requires a facility to save information in a file on the server for a user. The initial plan is to allow one file per CSCI version per user. In this way, the user and the CSCI version determine the name of the file. Having a fixed name avoids the need for a full interface for creating, naming and managing these files on the server. In the future, it would be reasonably straightforward to add such an interface to allow the user to manage their files.

5.6.1 User Preferences

Given the fact that there will be a place on the server that stores user data between sessions, it will be possible to use that place for other purposes. For example, user preference information could be stored in their server directory. Initially, the support for user preferences will be limited. Eventually users should be able to set preferences for items like the colors and fonts for aspects of the source listings (e.g., bold for keywords). The source listing styles will be implemented using style sheets. A first step to allowing the user to set their preferences would be to accept a user-defined style sheet that would override the system style sheet. Later, the system could provide an interface that would allow the user to choose colors and fonts interactively, and generate a style sheet for them.

A user could also choose a default software component to open when they sign on to the system. This information would be saved along with the other user preferences. It might even be possible to save some additional information about the previous session in order to restart with the same results and graph views visible.

5.7 Software Component Installation

When a new version of a software component is received there will be a process to install it in the system. This process will be automated as much as possible. The system administrator will need to create a new directory in the source repository for the new component, and install all the sources there. Once the sources are all in one place, the Ada program library needs to be created and all the sources compiled to generate the information and listings for analysis. Finally, once the library has been built without errors, the information about the CSCI and version needs to be registered in the Source Repository Configuration File so that the new component is available to users.

Initially, this process will probably consist of scripts that must be run on the Unix server. Eventually, it should be possible to provide a web interface for the installation process. Additional security would be needed for this interface, as the user would be modifying the data on the server, not just browsing it.

6. Future Enhancements

The architecture must be flexible and extensible to allow for new features. The following sections describe how some potential enhancements would fit into this architecture.

6.1 Additional Analysis

The ability to add additional analysis capabilities to this tool is critical. This architecture supports this need in two ways. First, new features can be added to the Xref engine. When a new feature is added to the engine, a new set of macros adding the feature to the menus will have to be released at the same time. Since this doesn't require any changes to the browser, the user will see the new feature as soon as it is available. Also, by isolating the changes to the macros, the generated source listings do not have to be regenerated. Second, new analysis tools can be added on the server. Like adding new features to the Xref engine, the macros would have to be updated. Furthermore, the dispatcher would have to be modified to recognize the new URLs and dispatch them to the appropriate tool. In both cases the output would probably still be HTML pages, possibly with new Java Applets embedded in them to support the new features. The goal in all cases would be to avoid requiring the user to install some application on their system to support the system.

6.2 Multiple Languages

Supporting a different language would require the implementation of a new set of scripts to implement menus and commands that are appropriate to the language. The filtering process would need to choose the appropriate set of scripts for the language when expanding the macros embedded in the listings. Language could easily be specified as a configuration parameter that is associated with a given CSCI.

The styles that are recognized by the system should be generic enough to apply to different languages. The styles will apply to entities like keywords, comments, string literals and identifiers. If an analyst chooses to have comments appear in brown italics, comments in every language should appear as brown italics.

To support another language, the xref engine would have to be replaced with an engine that provides comparable functionality for that language. Many of the functions supported by the xref engine, such as **find uses** will have similar commands in a different language. It should be possible to develop a standard API between the dispatcher and the "engine" that would allow the dispatcher to be language independent.

6.3 Configuration Management

In the current system, all the files associated with a software component version are expected to be in a directory that is specified in the configuration file. In the future the server functionality could be enhanced to support an interface to a configuration management system. Most CM systems have the concept of either a "sandbox", or a virtual file system. When the analyst chooses a CSCI version to work on, the system would need to check that version out of the CM system into a sandbox associated with the user. This would probably result in the use of less disk space on the server, as only CSCIs that were in use would be expanded into sandboxes. Other CSCI versions could be stored more efficiently by the CM system, either as source file "diffs", or as compressed versions.

6.4 Distributed Servers

In phase 2 a single central server where the source repository resides is assumed. When a large number of CSCI versions are being used by a large number of analysts, this arrangement could result in a performance bottleneck. Additionally, if the analysts are worldwide, analysts with internet connections closer to the Internet backbone will get better performance than those on the fringes.

A solution to this problem is to implement distributed servers for the repository. There would probably still be a single *master* server that holds the *master* copy of the repository. However, other servers, distributed geographically around the world would cache CSCI versions being worked on by local analysts. As far as the analyst was concerned their local server would appear to have everything. If they request a CSCI version that is not on the local machine the server would request it from the master server automatically. Obviously, the first time a CSCI version is accessed it would take somewhat longer to transfer the files, but subsequent accesses would be much faster. The local server would use a Least Recently Used algorithm to flush old CSCI versions out of its cache when necessary. The local server would also check automatically for updates to the source repository configuration file to present users with the latest information of what is available.

6.5 Development and Integration Phases

The main difference between supporting V&V and the development or integration phases of the software lifecycle is the fact that the sources are changing. To support the earlier development cycles two new interfaces would be needed. First, there would need to be an interface to the Configuration Management system that would allow users to check out, modify and return individual files. Second, there would need to be an interface to the compile and build functions of the compiler.

The phase 2 implementation of the tool takes advantage of the fact that the sources are static to pregenerate the HTML listings. If the users have the ability to modify the source files, then the interface would have to be able to handle changing files. Either the listings would have to be regenerated when the file is checked in, or the HTML listing would have to be generated on the fly from the current source when a user requests it.

7. Abbreviations & Acronyms

Abbreviation/Acronym	**Definition**
API	Application Programming Interface
CM	Configuration Management
CORBA	Common Object Request Broker Architecture
CSCI	Computer Software Configuration Item
CSS	Cascading Style Sheets
DID	Data Item Description
GUI	Graphical User Interface
HTML	Hyper Text Markup Language
HTTP	Hyper Text Transport Protocol
ICD	Interface Control Document
NSAPI	Netscape Application Programming Interface
PC	Personal Computer
SIAT	Software Interface Analysis Tool
URL	Universal Resource Locator
V&V	Verification & Validation
W3C	WorldWide Web Consortium
WAI	Web Application Interface